

# Uso de Git desde Eclipse

## Objetivo de este documento

Este documento contiene una serie de notas que describen diversos conceptos sobre el uso de Git y cómo se maneja Git desde Eclipse.

## Historia

V1.0, J. Tuya, Septiembre de 2016. Inicial

V1.1, J. Tuya, Febrero de 2017. Revisión general. Limpieza de Ramas. Workflow

V1.2, J. Tuya, Septiembre de 2017. Revisión general

V1.3, J. Tuya, Febrero de 2018. Añade precauciones con los finales de línea

V1.4, J. Tuya, Febrero de 2019. Comentarios adicionales sobre merge y rebase

V1.4, J. Tuya, Septiembre de 2019. Revisión general y más comentarios/enlaces sobre rebase

## Contenido

1	Conceptos iniciales.....	2
2	Situación inicial .....	2
2.1	Disponer de un repositorio remoto .....	2
2.2	Instalar Git en Eclipse.....	2
2.3	Disponer de un proyecto para la carga inicial.....	3
3	Conectar proyectos de Eclipse con Git .....	3
3.1	Cargar el repositorio desde un proyecto existente .....	3
3.2	Cargar un proyecto existente en el repositorio hacia Eclipse.....	6
4	Realizando cambios desde un único usuario .....	7
4.1	Commit y Push .....	7
4.2	Examinando los cambios realizados .....	8
4.3	El índice de Git .....	8
5	Cambios de varios usuarios en la misma rama .....	9
5.1	Commit y Push concurrentes .....	10
5.2	Actualización con los cambios de otro usuario (Pull) .....	10
5.3	Resumen de todos los comandos .....	10
6	Cambios de varios usuarios utilizando distintas ramas .....	11
6.1	Trabajando en una rama independiente (checkout) .....	11
6.2	Juntando los cambios de diferentes ramas .....	12
6.3	Solución de conflictos en merge .....	15
7	Manteniendo limpio y ordenado el historial de cambios .....	17
7.1	Ordenando los commit: Rebase.....	18
7.2	Agrupando commits: Interactive Rebase.....	18
7.3	Merge squash.....	20
7.4	Precauciones al alterar la historia de las ramas.....	21
7.5	Merge vs. Rebase .....	21
8	Otras posibilidades y características .....	22
9	Organización del workflow .....	23

# 1 Conceptos iniciales

Git es una herramienta que facilita la coordinación entre equipos de desarrollo, manteniendo el código fuente en sus diferentes versiones y proporcionando funciones, como la incorporación de cambios realizados concurrentemente por varios usuarios. Aunque se puede usar localmente, cuando se usa Git se mantienen tres lugares diferentes:

- Repositorio remoto: almacenado en un servidor que puede ser propio o un servicio como GitHub o BitBucket (en este documento se utilizará BitBucket).
- Repositorio local: almacenado en el equipo de trabajo que mantiene una copia parcialmente sincronizada con el repositorio remoto.
- Espacio de trabajo local (workspace): donde se realizan los cambios en el código.

Se accede a los repositorios git mediante línea de comandos o herramientas específicas más gráficas. Entre ellas se encuentra un plugin de Eclipse que será el utilizado aquí.

El flujo de trabajo más básico consiste en mover el código entre los tres repositorios anteriores:

- Los cambios realizados localmente son enviados al repositorio local utilizando el comando **commit**.
- Estos cambios son enviados al remoto utilizando el comando **push**.

Pero ocurren muchas situaciones en el trabajo de proyecto que complican la situación. Por ello Git proporciona un número relativamente amplio de comandos para la gestión del código, lo que unido a la existencia de dos repositorios diferentes puede dificultar el uso inicial si no se tienen claros algunos conceptos y procedimientos.

En esta guía se proporcionan los conceptos y procedimientos básicos y detalles para usar Git en un proyecto desde Eclipse. Para diferenciar los conceptos de los procedimientos desde eclipse, los conceptos son resaltados en color azul.

**Nota: lo visualizado en las pantallas puede diferir ligeramente dependiendo de la versión de eclipse (este documento se ha realizado usando las versiones luna y mars)**

## 2 Situación inicial

Debemos disponer inicialmente de un repositorio remoto Git alojado en un servicio externo, un proyecto en Eclipse y el plugin EGit instalado, como se describe a continuación.

### 2.1 Disponer de un repositorio remoto

Primero se necesita un repositorio Git, p.e. en BitBucket (<http://bitbucket.org/>):

- Crear cuenta y un repositorio en blanco, que llamaremos *demo1*.
- La URI del repositorio es de la forma:  
<https://<usuario>@bitbucket.org/<usuario>/<repositorio>.git>
- Posteriormente podremos crear un equipo y asignar miembros al mismo

### 2.2 Instalar Git en Eclipse

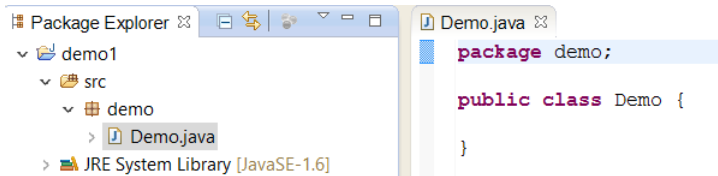
Eclipse incorpora EGit que es una versión del cliente Git para uso desde el propio workspace. La mayoría de las operaciones se pueden realizar desde la perspectiva Java, aunque incorpora otra perspectiva específica para Git. En caso de que no esté instalado, hay que instalarlo de la siguiente forma

- Seleccionar **Help->Install New Software**

- Añadir una url: <http://download.eclipse.org/egit/updates>
- Y seleccionar **Eclipse Git Team Provider** y **JGit**

### 2.3 Disponer de un proyecto para la carga inicial

Tenemos un proyecto en eclipse: *demo1* que es el que queremos compartir enviándolo a un repositorio Git. Partiremos del siguiente:



## 3 Conectar proyectos de Eclipse con Git

Existen dos casos que se mostrarán a continuación:

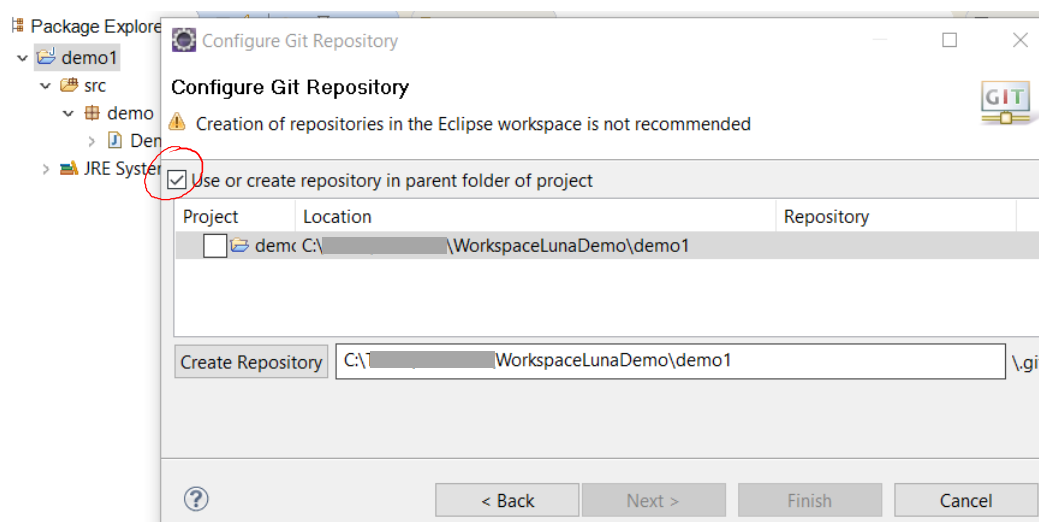
- Partir de un proyecto existente, asociarlo y cargarlo en Git (**Share**)
- Partir de un workspace en blanco, asociar y cargar un proyecto que ya existe en Git (**Import**)

El repositorio local se almacena en una carpeta del workspace de Eclipse denominada *.git*.

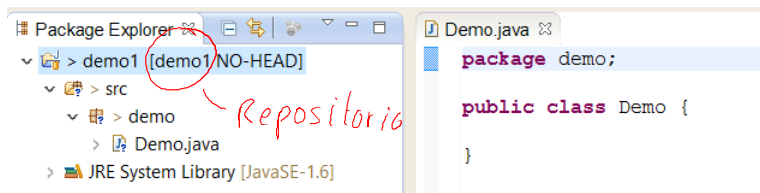
### 3.1 Cargar el repositorio desde un proyecto existente

Partiremos de un workspace con el proyecto indicado en 2.3. El proceso de carga de un proyecto nuevo en el repositorio Git consiste en asociar los ficheros del proyecto a un repositorio local que a su vez esté asociado al repositorio remoto. Es un poco largo, pero solo se realiza una vez en la vida del proyecto. Una forma de hacerlo es desde la perspectiva Java, de la siguiente forma:

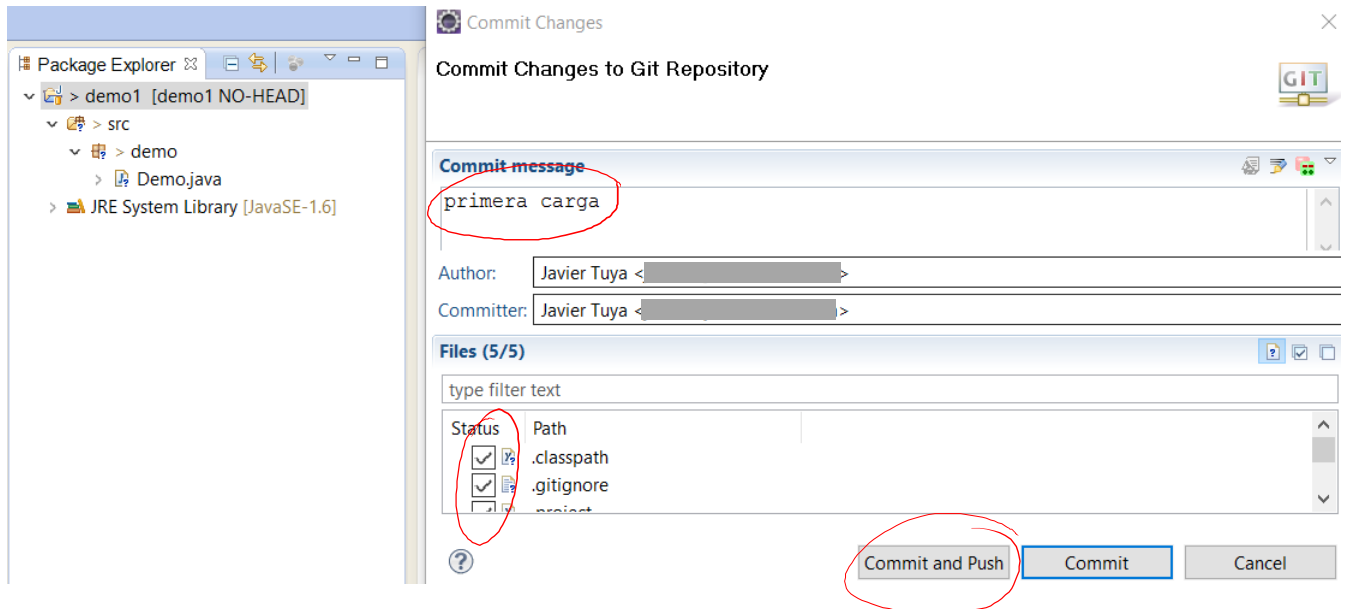
1) Seleccionar la raíz del proyecto y crear el repositorio local asociado (**Team->Share Project->Git**). Hay que especificar donde se guardará el repositorio local, que puede estar en cualquier lugar o en la misma raíz del proyecto (será una subcarpeta de nombre *.git*). En este último caso, se marcará la opción "Use or create repository in parent project folder" -> click en **Create Repository -> Finish**):



Ahora ve el proyecto asociado a un repositorio local, pero los datos están pendientes de guardar (se ven marcas de interrogante en los iconos):



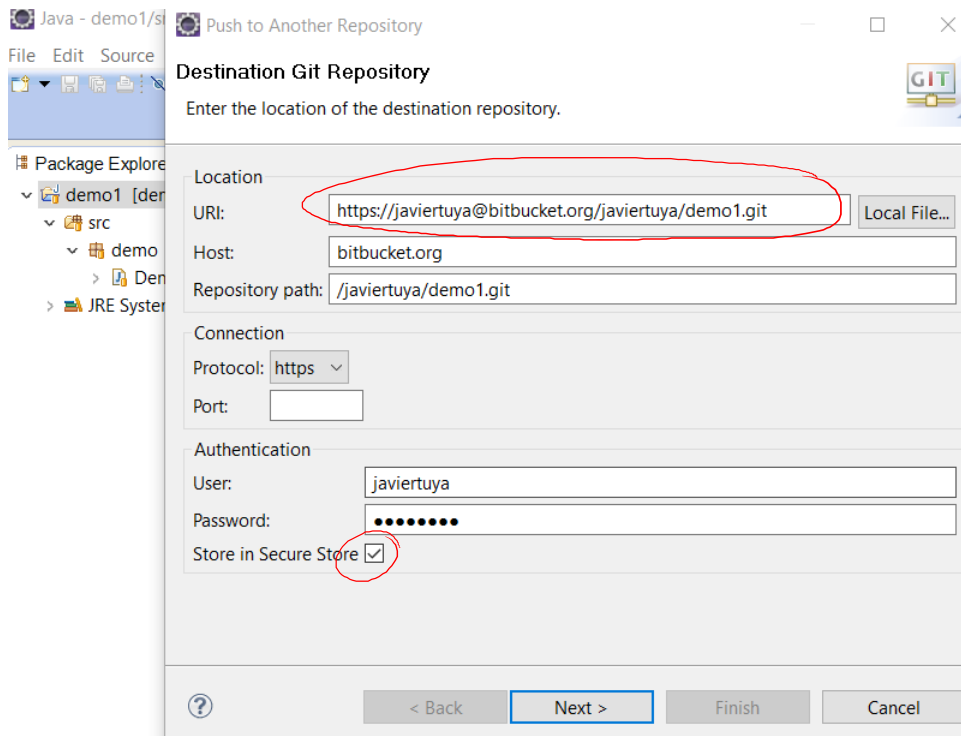
2) Iniciar la carga en el repositorio (**Team->Commit**). Se inicia el diálogo en el que seleccionamos todos los archivos y un mensaje descriptivo del cambio realizado (en este caso, primera carga):



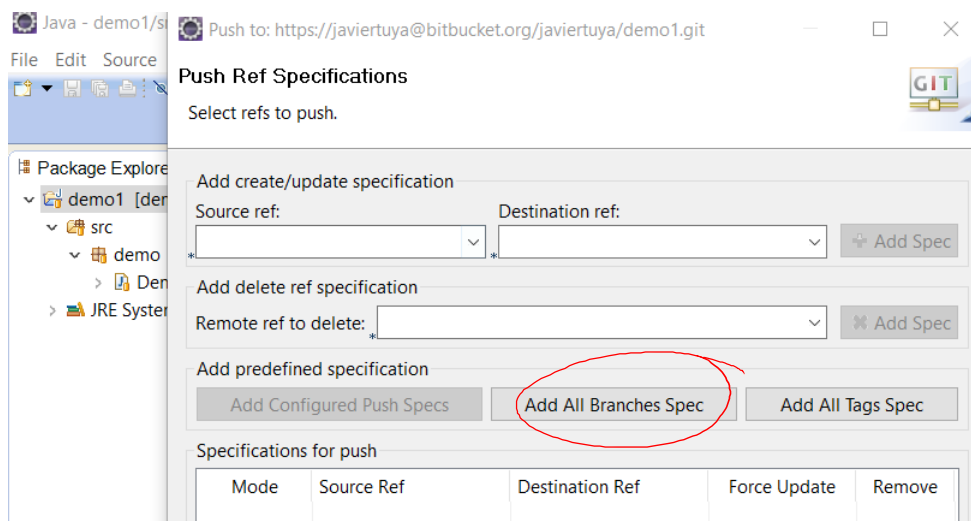
Hay dos opciones:

- **Commit** guardará los datos en el repositorio local solamente.
- **Commit and Push** los enviará también al repositorio remoto, que es lo que queremos hacer.

3) Cargamos los datos en el repositorio remoto (**Commit and Push**), indicando sus datos (introduciendo la URI el resto de datos salvo el password se cumplimentarán por defecto). Indicamos también que guarde los datos de acceso para no tener que repetirlos:

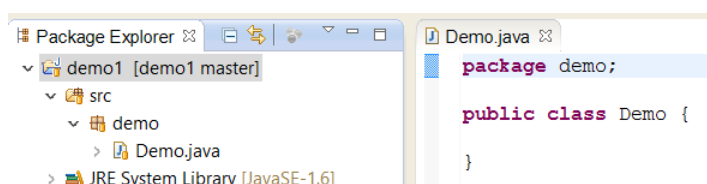


4) Todavía falta una acción adicional en el siguiente diálogo que establece cómo se enlazarán repositorio local y remoto:



Solamente hay que especificar cómo se enlazarán repositorio local y remoto, y finalmente **Add All Branches Spec -> Next -> Finish -> OK**.

5) Comprobaciones adicionales. El proyecto ya no mostrará cambios pendientes, quedando de la siguiente forma:



Se puede comprobar que se ha hecho una actualización en el repositorio remoto desde bitbucket seleccionando el repositorio y "Commits"

Java demo1

ACTIONS

- Clone
- Create branch
- Create pull request
- Compare
- Fork

NAVIGATION

- Overview
- Source
- Commits

Javier Tuya / demo1

## Commits

All branches ▾

Author	Commit	Message
Javier Tuya	860c21a	carga inicial

Seleccionando el id de uno de ellos se ven los archivos que han cambiado:

Java demo1

ACTIONS

- Clone
- Create branch
- Create pull request
- Compare
- Fork

NAVIGATION

- Overview
- Source
- Commits
- Branches
- Pull requests
- Downloads
- Settings

Javier Tuya / demo1

## Commits

Javier Tuya committed 860c21a 2 minutes ago

carga inicial

Comments (0)

What do you want to say?

Files changed (5)

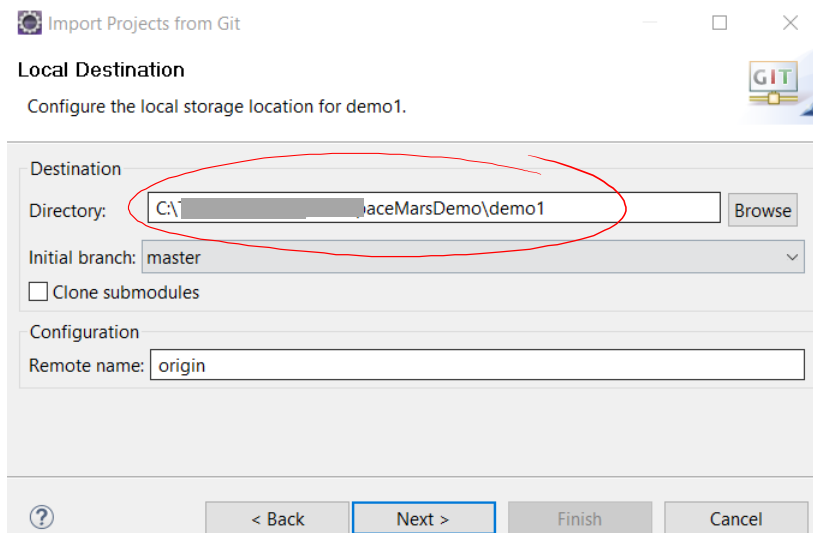
+6	-0	.classpath
+1	-0	.gitignore
+17	-0	.project
+11	-0	.settings/org.eclipse.jdt.core.prefs
+5	-0	src/demo/Demo.java

### 3.2 Cargar un proyecto existente en el repositorio hacia Eclipse

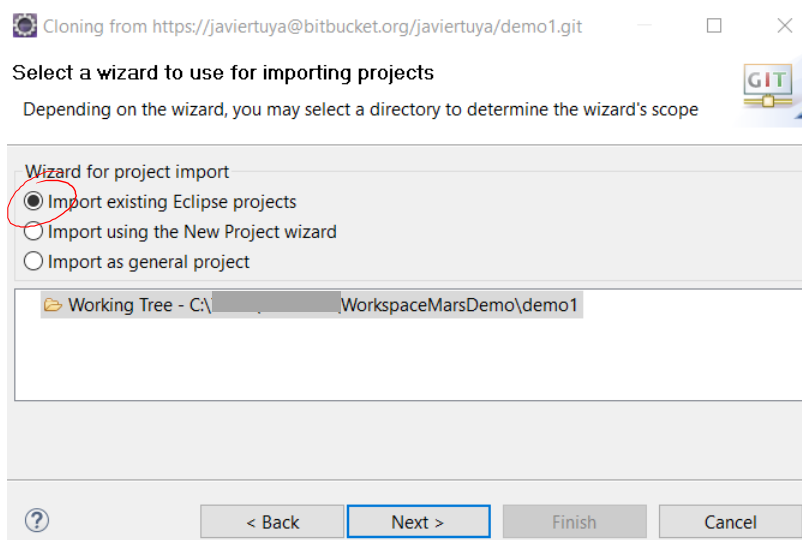
Lo más habitual es que ya se tenga un proyecto en el repositorio Git remoto, y queremos cargar este en nuestro entorno de desarrollo empezar a trabajar:

Desde un workspace importar el repositorio Git en un nuevo proyecto (**File->Import->Git->Projects from Git->Next->Clone Uri->Next**).

Se abre el dialogo para indicar los datos de acceso al repositorio a importar. Introducir estos datos -> **Next -> Next**. En el diálogo que aparece hay que especificar la carpeta donde se encontrará el proyecto:



**Next** abre otro diálogo, Indicar que se importarán todos los proyectos del repositorio



Finalmente, **Next -> Finish** y ya se dispone del proyecto asociado en este workspace.

## 4 Realizando cambios desde un único usuario

Repasando algunos comandos antes usados: **Commit** envía cambios realizados en el workspace al repositorio local y **Push** los envía al repositorio remoto. Si hay archivos que no se desea que se mantengan en el workspace pero que no sean enviados al repositorio, editar el archivo `.gitignore` para especificar los que se deseen. Desde Eclipse también se puede hacer simultáneamente **Commit and Push** como se indicó anteriormente. Ahora se partirá de un proyecto asociado a Git como el anterior y vamos a realizar cambios:

### 4.1 Commit y Push

Añadimos algunas líneas de código al método `main`:

```

Demo.java
1 package demo;
2 public class Demo {
3     public static void main(String[] args) {
4         System.out.println("linea 1");
5         System.out.println("linea 1");
6         System.out.println("linea 1");
7     }
8 }

```

Y actualizamos el repositorio remoto (**Team->Commit->indicar un comentario->Commit and Push**). Siempre que se realiza un commit es obligatorio indicar un comentario breve que condense en una línea los cambios realizados, para facilitar el conocimiento de lo que ha ocurrido desde el historial.

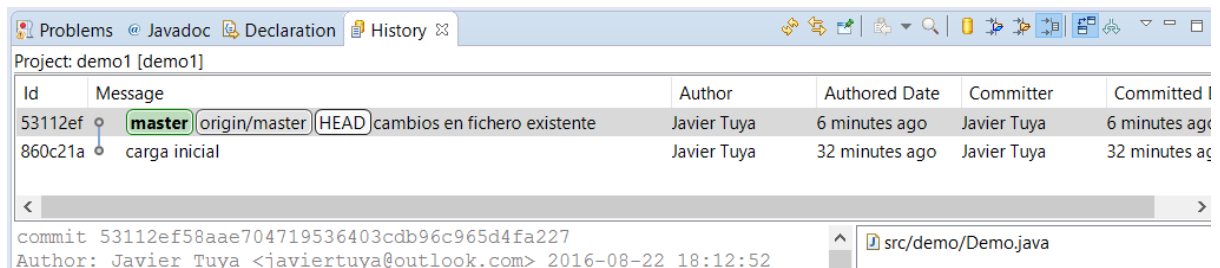
## 4.2 Examinando los cambios realizados

Desde bitbucket se ven los nuevos cambios

Author	Commit	Message
 Javier Tuyá	53112ef	cambios en fichero existente
 Javier Tuyá	860c21a	carga inicial

Notar que Commit se usa muchas veces como sustantivo, para referirse a los cambios realizados en un momento dado. Cada uno se identifica con un id único.

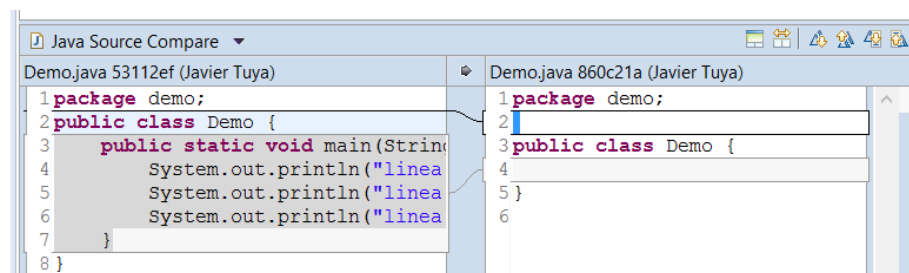
También se puede ver la historia desde Eclipse tras seleccionar la raíz del proyecto (**Team->Show in History**):



Id	Message	Author	Authored Date	Committer	Committed I
53112ef	cambios en fichero existente	Javier Tuyá	6 minutes ago	Javier Tuyá	6 minutes ago
860c21a	carga inicial	Javier Tuyá	32 minutes ago	Javier Tuyá	32 minutes ago

commit 53112ef58aae704719536403cdb96c965d4fa227  
 Author: Javier Tuyá <javiertuya@outlook.com> 2016-08-22 18:12:52

En la parte inferior se ven los archivos que han cambiado de una versión a otra (en este caso, *Demo.java*). Con doble click en uno se pasa a la pantalla de comparación de archivos.

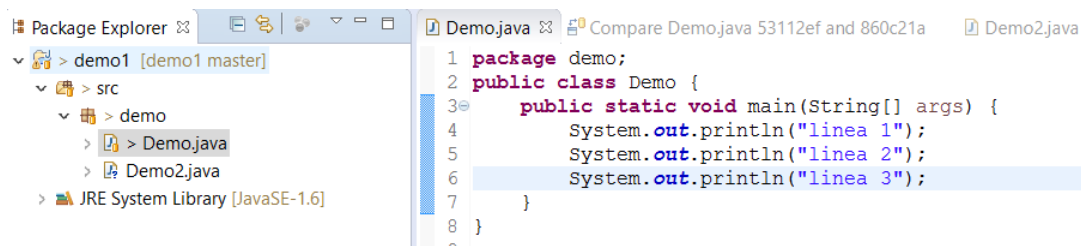


Si ha habido cambios en el proyecto sin guardar en el repositorio, lo más habitual es necesitar compararlo con la última versión del repositorio (**Compare With -> HEAD Revision**).

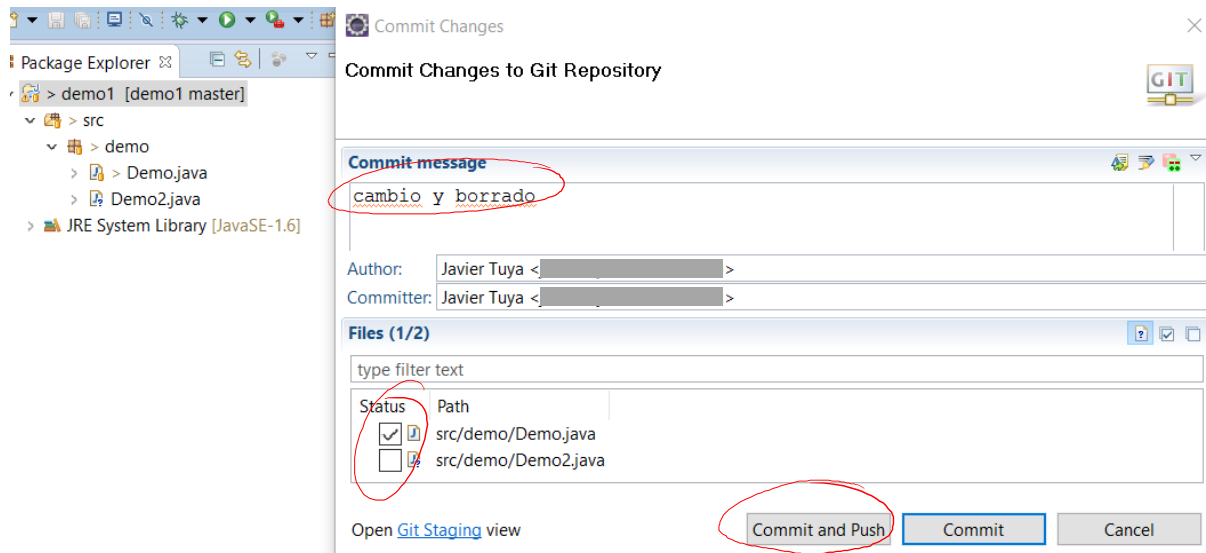
## 4.3 El índice de Git

Hacemos un cambio en *Demo.java* y creamos otra Clase *Demo2.java*:



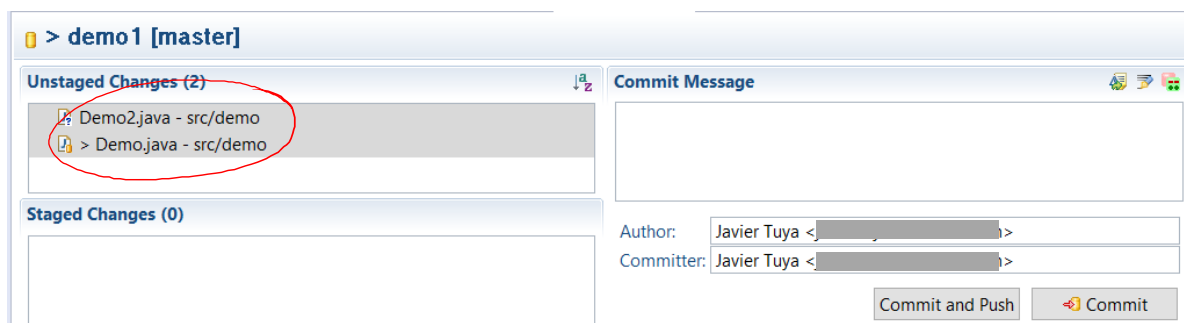


Al hacer **Team -> Commit** se ve lo siguiente:



Nótese que en la parte inferior solamente aparecen marcados los archivos que han cambiado, no los nuevos. Cuando se crean (y se borran) archivos, hay que indicar explícitamente que se van a añadir al índice de git.

Podemos hacerlo desde esta misma pantalla seleccionando los nuevos archivos, o abrir la **Git Staging view**, seleccionando archivos, click derecho y **Add to Index**:



**Importante:** Los cambios que son enviados a un repositorio local con commit no son todos los que hemos hecho en el workspace, sino solamente los han sido previamente añadidos al índice de Git. Es importante no olvidar esto para evitar hacer commits con un proyecto incompleto. Luego el flujo es el siguiente:

Archivos en el workspace – índice de Git – Repositorio local – Repositorio remoto

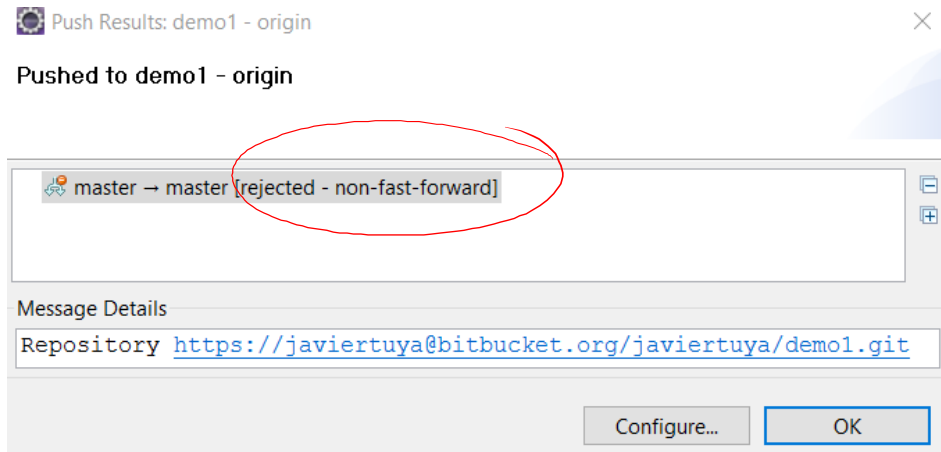
## 5 Cambios de varios usuarios en la misma rama

En la sección anterior solamente un usuario cambio los archivos. Pero, ¿qué ocurre cuando otro usuario incorpora cambios antes de que lo hagamos nosotros?

## 5.1 Commit y Push concurrentes

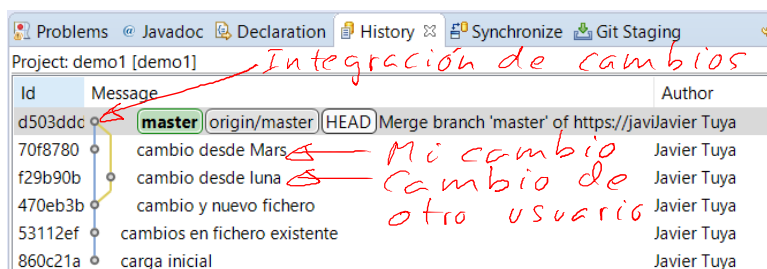
Supongamos que, tras la situación anterior empezamos a hacer cambios. Pero simultáneamente, otro usuario incorpora un cambio, y hace el commit y push.

Cuando finalizamos de incorporar los cambios hacemos commit y push. Pero este cambio es rechazado porque lo habíamos hecho respecto de una versión que no es la última (puesto que el otro usuario introdujo una nueva versión). *Se dice que nuestro commit no es fast-forward porque trata de incorporar cambios que fueron realizados partiendo de una situación del repositorio anterior a la actual:*



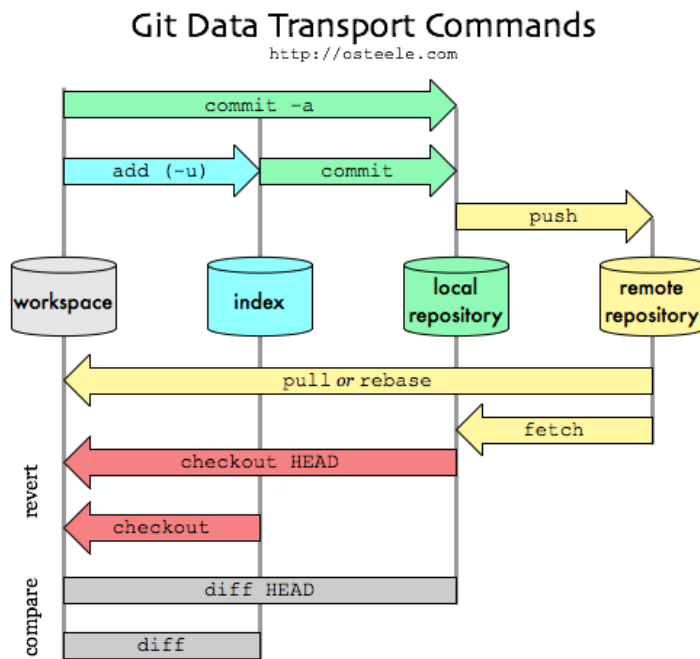
## 5.2 Actualización con los cambios de otro usuario (Pull)

El comando **Fetch** trae desde un repositorio al workspace los últimos cambios que se han realizado en el repositorio (normalmente por otros usuarios). Pero si nosotros también hemos hecho cambios necesitamos mezclar nuestros cambios con los de los otros. Para ello se utiliza **Pull**, que realiza una combinación de **Fetch** (traer los cambios de la última versión) seguido de un **Merge** (mezclarlos con los cambios locales). En nuestro caso, desde Eclipse haríamos **Team->Pull**. Todo el proceso se ve en el historial (**Team->Show in History**).



## 5.3 Resumen de todos los comandos

Se puede ver en la siguiente figura:



## 6 Cambios de varios usuarios utilizando distintas ramas

La anterior forma de trabajo puede suponer bastante interferencia entre diferentes usuarios, haciendo difícil saber cuándo se dispone de la última versión y la solución de conflictos que puedan ocurrir, por lo que lo habitual será trabajar utilizando diferentes ramas. Esta será la forma de trabajo en las prácticas.

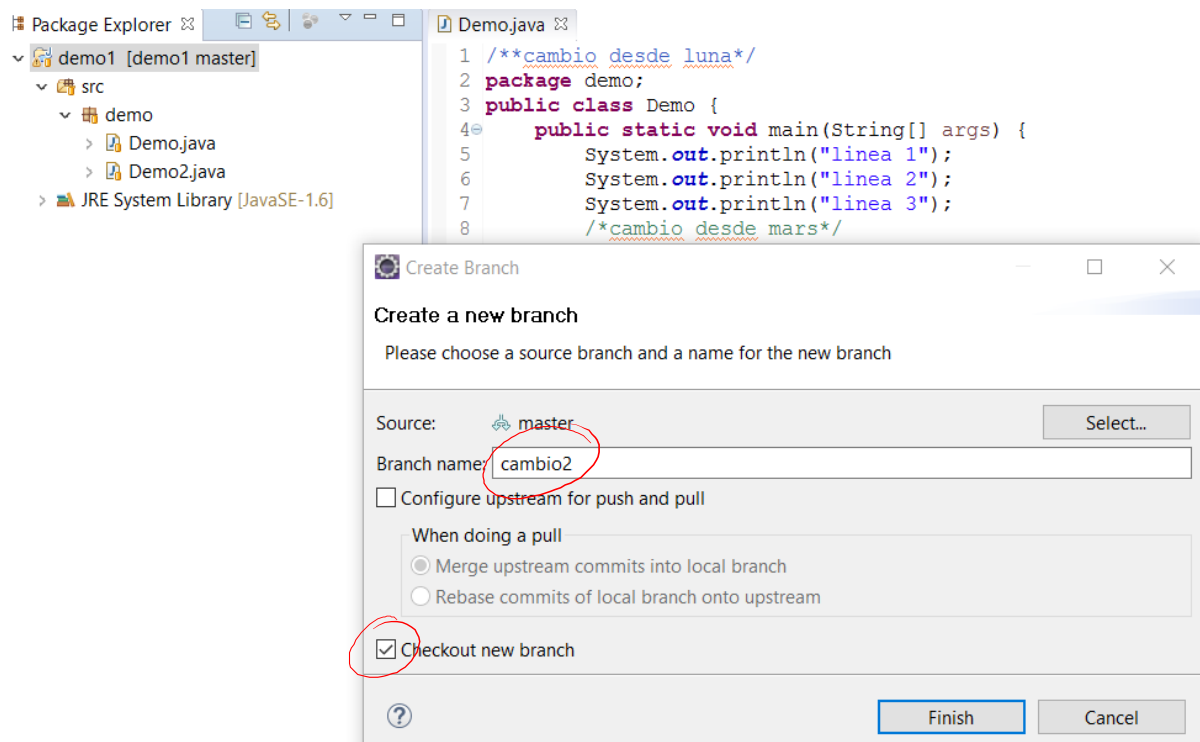
En un esquema simple usando ramas, la rama principal (*master*) será la usada para la versión integrada con las contribuciones de todos los usuarios, cada cambio realizado por un usuario se hará en una rama diferente a la que se dará un nombre (es la rama en la que trabajamos), eliminando interferencias entre el trabajo de varios usuarios.

Cada una de las ramas mantiene la historia temporal de todos los commits que se han realizado en ella. Se dice que una rama es privada cuando solo está en el repositorio local (nunca se ha hecho push), de lo contrario se dice que la rama es pública.

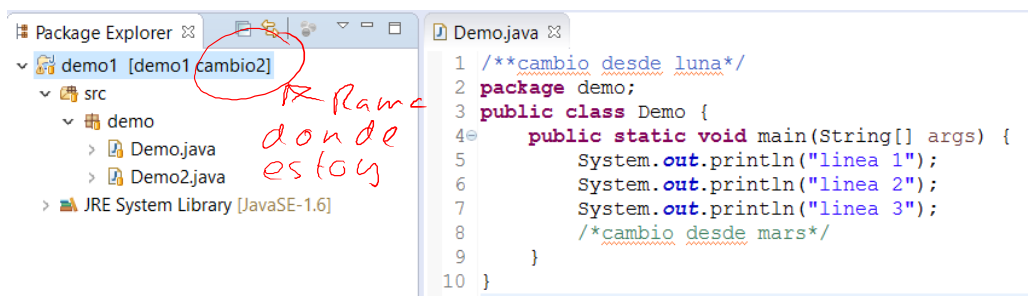
### 6.1 Trabajando en una rama independiente (checkout)

Continuamos haciendo más cambios, pero en este caso, sobre una rama diferente, de la siguiente forma:

1) Creamos la nueva rama *cambio2* que contendrá los mismos archivos que la rama en la que estamos (*master*) y hacemos el *checkout* para situarnos en ella (**Team -> Switch To -> New Branch**). Indicar el nombre de la rama y asegurarse de marcar la casilla **Checkout new Branch**:



Tras **Finish**, en la raíz del proyecto se ve que se está trabajando en otra rama (*cambio2*) distinta de *master*



2) Realizar los cambios en esta rama. Tras hacer un commit al repositorio local, veremos estos en el historial:

Problems @ Javadoc Declaration History Synchronize Git Staging		
Project: demo1 [demo1]		
Id	Message	Author
c38dc40	<b>cambio2</b> HEAD cambio 2 desde mars en rama cambio2	Javier Tuya
c688842	<b>master</b> origin/master estado inicial antes de merge	Javier Tuya

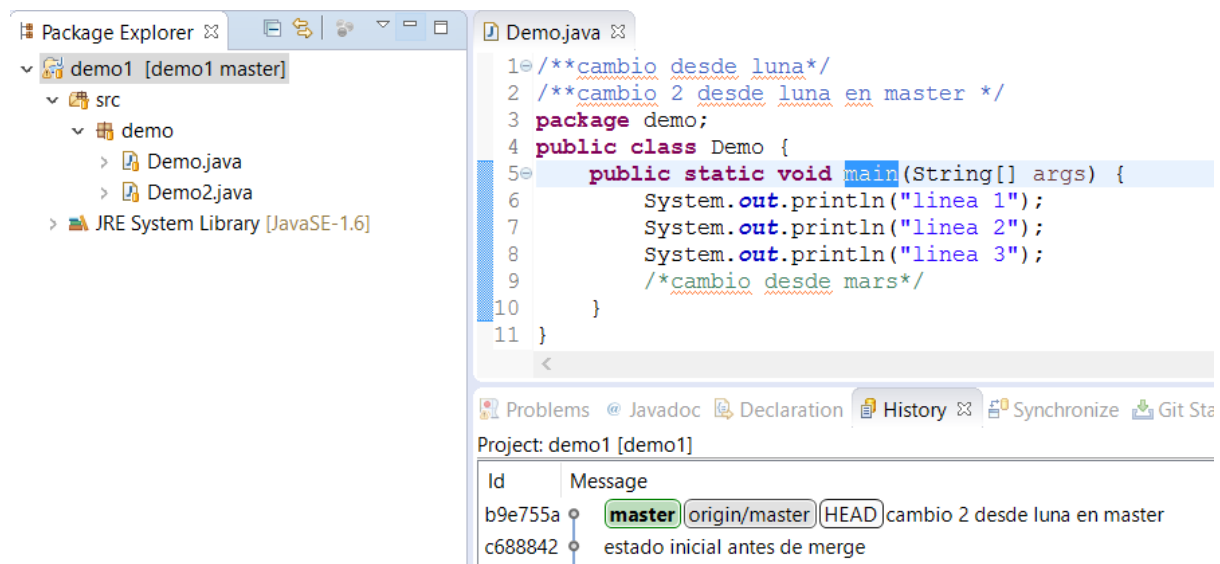
**Importante:** Este tipo de ramas no son públicas, ya que no se publican en el repositorio remoto (salvo que hagamos push de la rama explícitamente). Sin embargo, todos los commit que hagamos se verán finalmente en el repositorio cuando hagamos un merge (como se ha indicado anteriormente). Por ello sigue siendo esencial describir bien los cambios. Aunque es aconsejable hacer commits con cierta frecuencia, no se debe usar Git como copia de seguridad haciendo commit y push en momentos aleatorios del ciclo de vida de un cambio.

## 6.2 Juntando los cambios de diferentes ramas

La operación de juntar (o mezclar) los cambios realizados en dos ramas se denomina *merge*. Ahora vamos a integrar los cambios realizados en la rama *cambio2* en *master*. Pero supongamos la situación

habitual que desde que hemos creado la rama *cambio2* en la que hemos trabajado, otros usuarios han actualizado *master* en el repositorio remoto con uno o varios cambios. Procederemos de la siguiente forma:

1) Nos aseguramos de que en *master* tenemos la última versión con los cambios de otros usuarios: Cambiamos a la rama *master* (**Team -> Switch to -> master**), lo que causará que en nuestro espacio de trabajo todos los archivos se actualicen para corresponderse con los de esta rama. A continuación, actualizamos *master* con los cambios de los otros usuarios (**Team -> Pull**), con lo que veremos el cambio en el código y en la historia:

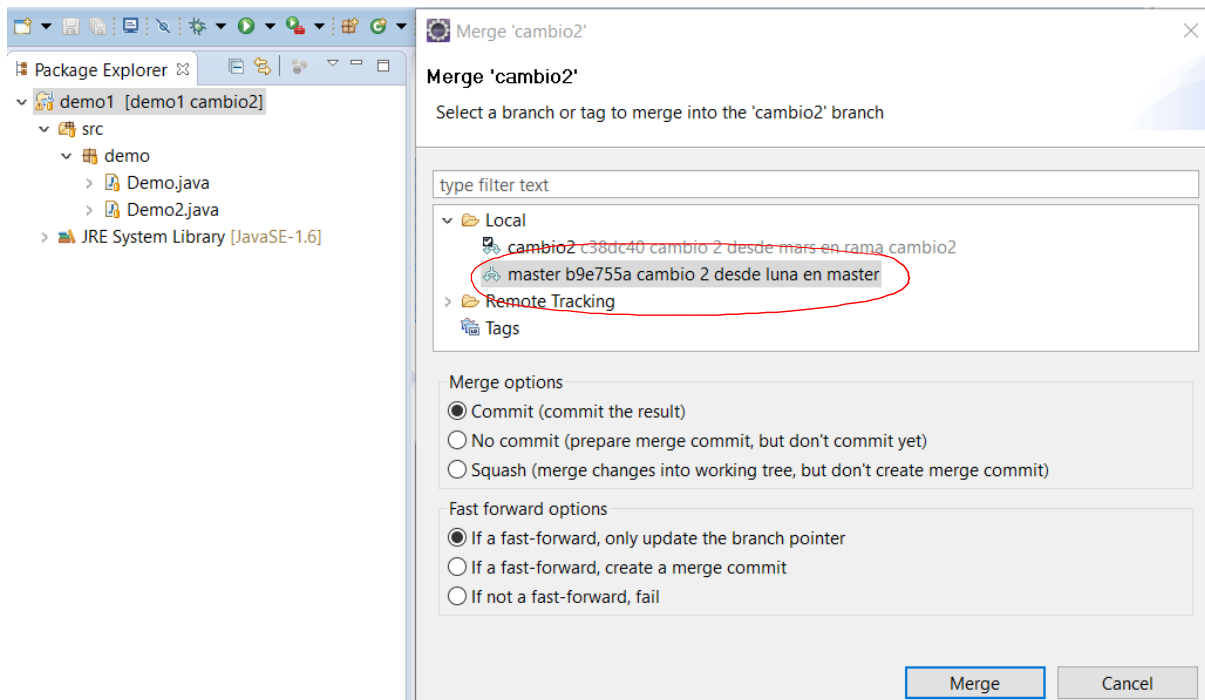


2) Desde aquí podríamos ya realizar el Merge para juntar con los cambios de la rama *cambio2*.

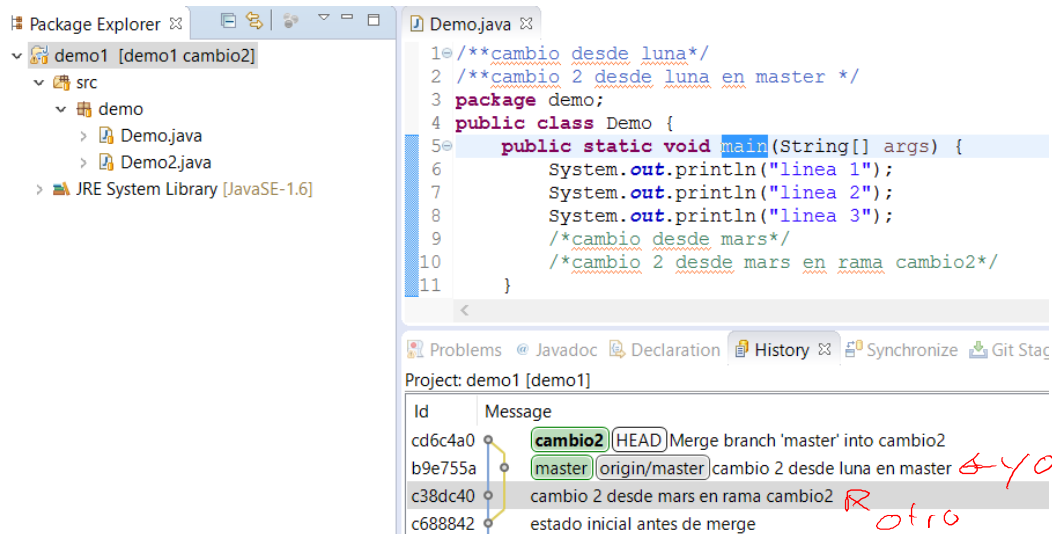
Pero pueden existir conflictos al realizar un merge (p.e. si varios usuarios han cambiado la misma parte de un archivo) cuya resolución se tratará más adelante. Además, siempre debemos mantener la rama *master* lo más libre posible de versiones temporales o conflictivas.

Por ello, en vez de proceder como se ha indicado, un procedimiento más seguro es integrar primero los cambios que otros hayan podido realizar en *master* en la rama de trabajo, y tras solucionar en la rama de trabajo los problemas o conflictos que puedan aparecer, integrar estos cambios en *master*. En este ejemplo, realizaremos primero la integración en *cambio2*:

Cambiamos a la rama *cambio2* (**Team -> Switch to -> cambio2**) y a continuación hacemos merge con el contenido de *master* (**Team -> Merge**). Seleccionamos la rama *master* en el diálogo que nos aparece.

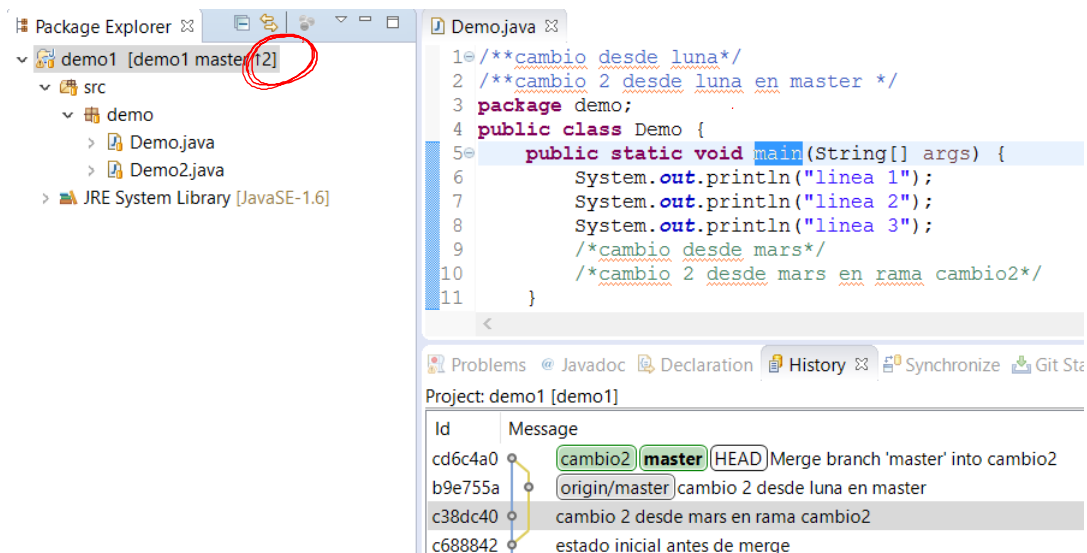


Ya tenemos en *cambio2* nuestros cambios más los que otros habían hecho simultáneamente en master:



3) El hecho de que Git consiga hacer un merge sin conflictos, o habiendo conflictos, los solucionemos, no implica que todo en nuestra rama esté bien, puesto que el resultado puede no funcionar correctamente o incluso no compilar. Por ello, antes de enviar a master, no debemos olvidarnos de probar la integración de nuestros cambios con los realizados por los otros usuarios. En caso de que haya problemas, realizar los cambios necesarios hasta solucionarlos, hacer un commit y repetir el proceso de integración de las ramas.

4) Tras verificar que todo está correcto en nuestra rama privada (*cambio2*) y hemos hecho el commit, ya podemos integrarlo en *master* con seguridad. Volvemos a *master* (Team -> Switch to -> master) y realizamos un merge con el contenido actual de la rama *cambio2* (Team -> Merge -> cambio2):



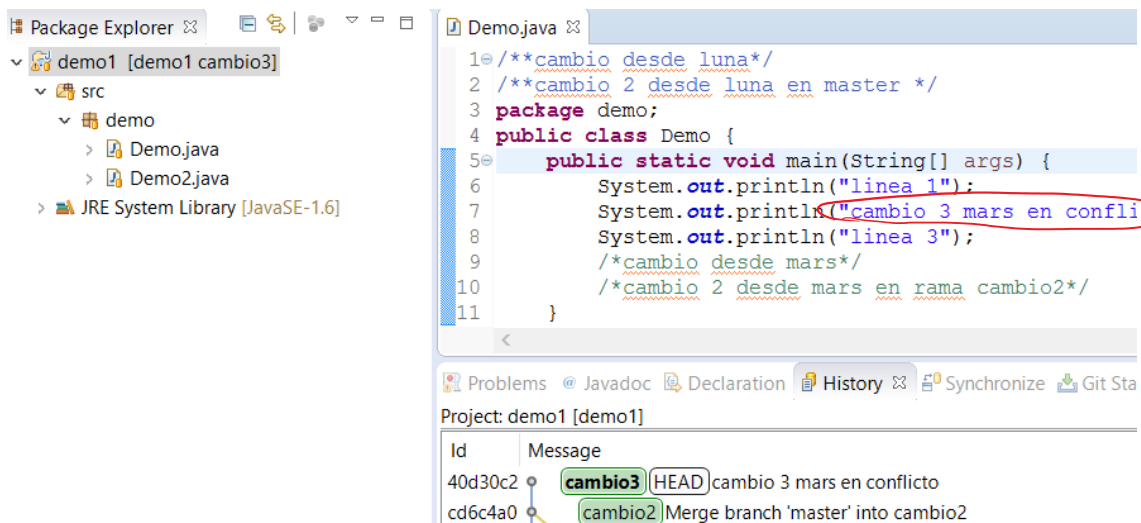
5) No debemos olvidarnos que los cambios están en el repositorio master local y están pendientes de enviar al remoto (se ve una flecha en la raíz del proyecto que lo indica), por lo que debemos finalizar con un push (**Team->Push branch 'master'**).

Recordar que cuando trabajamos en una rama y vamos a master para realizar este tipo de operaciones, debemos hacer un pull en master para actualizar el repositorio local con los últimos cambios del remoto.

### 6.3 Solución de conflictos en merge

La operación merge realizada anteriormente incorporó cambios de varios usuarios que no entraron en conflicto. Aunque estaban en el mismo fichero, estaban en diferentes líneas, por lo que Git pudo integrar los cambios. Tras realizar las pruebas todo ha ido bien y se ha podido continuar integrando los cambios en la rama master.

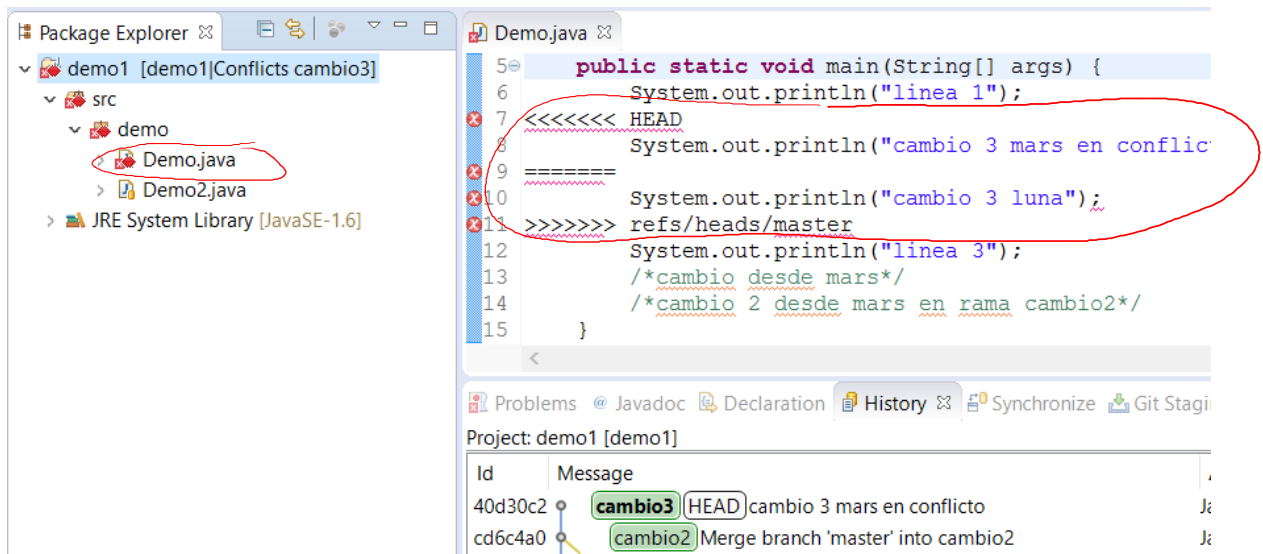
Supongamos que el cambio que se realiza es diferente: Creamos una rama *cambio3* en la que modificamos el texto del segundo println y hacemos commit:



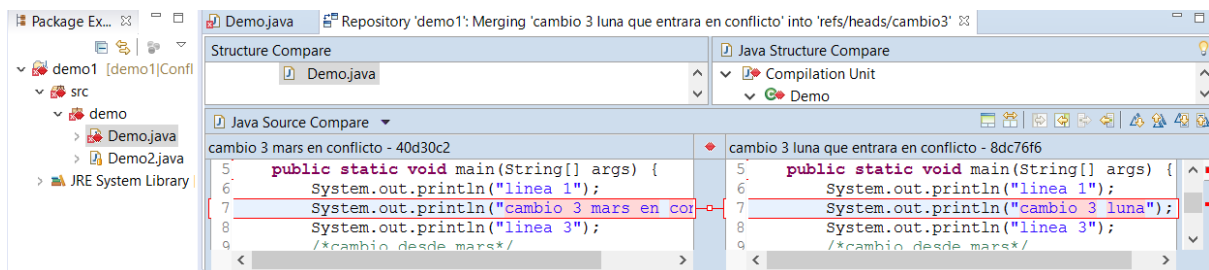
Mientras tanto, otro usuario ha integrado otros cambios en la rama **master** del repositorio remoto que también cambiaban la misma línea que hemos cambiado nosotros.

Para juntar los cambios seguimos el procedimiento de la sección anterior: vamos a *master* y actualizamos con los cambios de otros usuarios, volvemos a *cambio3* y hacemos merge con *master*. En

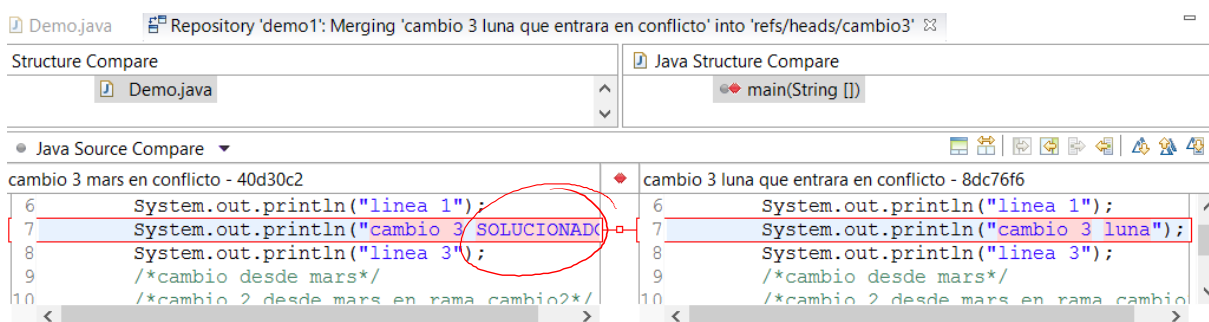
este caso aparece entonces un conflicto que es señalado en rojo en el explorador de paquetes, y en el código resultante nos aparecen las dos versiones en conflicto que tendremos que remplazar por la versión definitiva que solucione el conflicto.



También es posible resolver los conflictos más visualmente (seleccionar archivo en conflicto -> **Team -> Merge Tool**):



Por ejemplo, en este caso resolvemos el conflicto poniendo el texto definitivo en la ventana de la izquierda (la de la rama *cambio3* donde estoy)



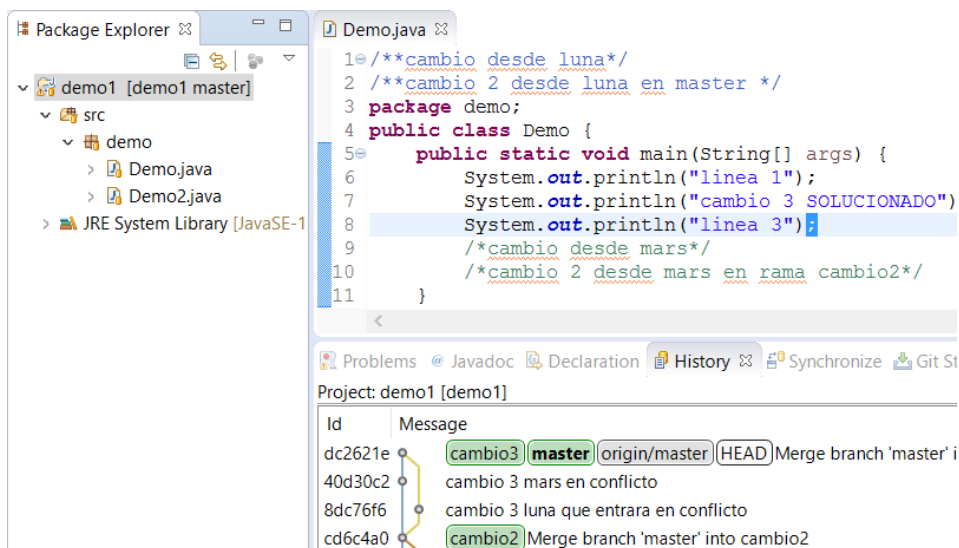
Para indicar que un conflicto está resuelto, añadimos el archivo tras solucionar los conflictos al índice de Git. En este caso añadimos *Demo.java* al índice de Git (**Team -> Add to Index**)

Cuando hay conflictos, el merge no finaliza la actualización de la rama, por lo que una vez resueltos todos los conflictos y añadidos al índice los archivos correspondientes, hay que hacer un commit:





Ahora ya podemos continuar el procedimiento cambiando a la rama *master*, realizando el merge con la rama *cambio3* y push.



## 7 Manteniendo limpio y ordenado el historial de cambios

Cuando trabajamos en una rama, aunque sea privada, todos los commits realizados serán integrados en cualquier rama en la que hagamos un merge. Supongamos que trabajamos en la implementación de la funcionalidad de una historia de usuario en una rama independiente de master. Es muy posible que hagamos diversos commits para reflejar cambios parciales para la implementación de la funcionalidad, o que hagamos commits para deshacer parte del camino andado, o para solucionar conflictos u otros problemas surgidos al hacer los commits. Integrar todos estos commits en master supondría “ensuciar” dicha rama (que quedaría con una gran cantidad de commits irrelevantes), con lo que se pierde la utilidad de la historia para conocer la evolución del proyecto. Lo ideal sería integrar solamente un commit en master que reflejase todo el trabajo realizado para implementar la historia de usuario.

Para ello existen diferentes técnicas y comandos Git que se comentan a continuación. En esencia:

- Rebase tiene como objetivo reordenar los commits.
- Squash tiene como objetivo juntar varios commits como si fueran uno solo.

## 7.1 Ordenando los commit: Rebase

Supongamos que hemos creado una rama *cambio-rebase*. Mientras nosotros estamos trabajando en la rama otro usuario hace un commit a master, luego hacemos un commit en nuestra rama, otro usuario hace otro commit a master finalmente nosotros un commit a nuestra rama.

Cuando hagamos un merge se reflejará el orden temporal en el que se hicieron todos los cambios. La siguiente es la situación de la rama *cambio-rebase* tras el merge con master, donde se aprecia que se entremezclan los cambios realizados por otros usuarios en master con los que hicimos en nuestra rama. Esto dificulta la comprensión de los cambios realizados en las ramas.

Id	Message
8467b4b	<b>cambio-rebase</b> HEAD Merge branch 'master' into cambio-rebase
1f418db	cambio 2 a cambio-rebase
9a912f7	<b>master</b> origin/master cambio 2 a master
e5ef172	cambio 1 a cambio-rebase
6183c9a	cambio 1 a master
95f4b5a	Reset a estado inicial

*Mis cambios* (pointing to 1f418db and e5ef172)

*Cambios de otros (master)* (pointing to 9a912f7 and 6183c9a)

Para ordenar mejor los commits existe la opción rebase (**Team -> Rebase**). **Rebase es una forma de hacer un merge en la que todos los commits que se han hecho en la rama que estamos integrando con la nuestra quedan ordenados al principio de nuestra rama.** En nuestro caso, una vez hecho el ultimo commit en nuestra rama, como siempre, iríamos a master y haríamos un pull para actualizar a la última versión, y de vuelta en nuestra rama haríamos rebase en vez de merge. Tras ello, en nuestra rama veremos primero todos los cambios hechos en master y luego todos los que hicimos en nuestra rama como se muestra a continuación.

Id	Message
c345166	<b>cambio-rebase</b> HEAD cambio 2 en cambio-rebase
0ab7914	cambio 1 en cambio-rebase
2225043	<b>master</b> origin/master cambio 2 en master
375b7d1	cambio 1 en master
1e546a6	Reset a estado inicial

*Mis cambios* (pointing to c345166 and 0ab7914)

*Cambios de otros (master)* (pointing to 2225043 and 375b7d1)

El efecto de Rebase es modificar la historia en una rama de forma que todos los cambios realizados en ella aparecen como si la rama se hubiera creado a partir del último commit realizado en la rama original, en lugar del commit desde el que se creó la rama.

De esta forma queda una historia más lineal, aunque no se ve en qué momento otros hicieron los cambios en master.

## 7.2 Agrupando commits: Interactive Rebase

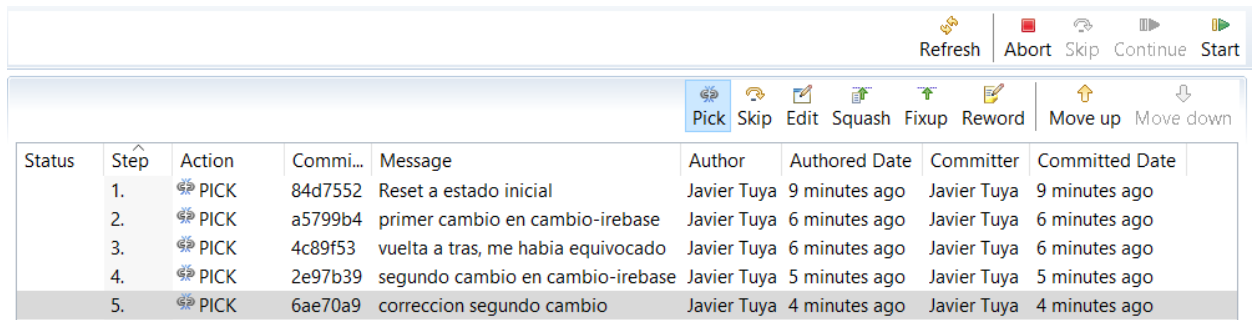
Rebase permite tener una historia más ordenada de las ramas, pero no evita que queden “sucias” con commits que pueden no ser de interés (temporales, correcciones, vueltas atrás, etc). La figura anterior refleja la historia en la rama *cambio-rebase* (que será la que se vea desde master hagamos merge con dicha rama). Supongamos que en nuestra rama (ahora llamada *cambio-irebase*), en vez de dos cambios, hemos realizado cuatro que no queremos que se reflejen finalmente en master. Además, mientras tanto, otros han hecho dos cambios en master). La siguiente sería la situación inicial de nuestra rama:

Id	Message
6ae70a9	<b>cambio-irebase</b> HEAD correccion segundo cambio
2e97b39	segundo cambio en cambio-irebase
4c89f53	vuelta a tras, me habia equivocado
a5799b4	primer cambio en cambio-irebase
84d7552	<b>master</b> origin/master Reset a estado inicial

No queremos que al final todos estos cambios sean enviados finalmente a master.

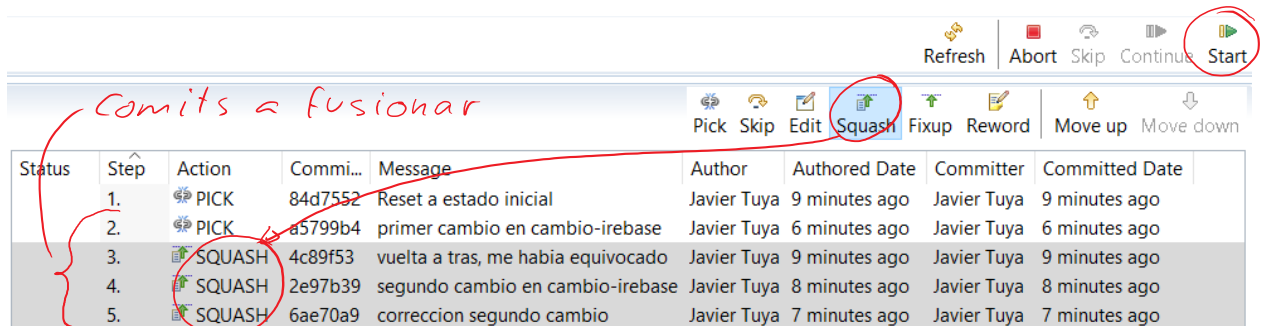
El **rebase interactivo** permite alterar la historia de nuestra rama utilizando una serie de operaciones, entre los cuales se encuentra **squash**, que permite juntar varios commits en uno solo.

En Eclipse, desde la historia (**Team -> Show in History**) nos situaríamos en un commit anterior a los que queremos juntar (p.e. el anterior a "Reset a estado inicial". Una vez en él seleccionamos **Team -> Interactive Rebase** que nos mostrará la historia (al revés, el más antiguo es el primero de la lista):



Status	Step	Action	Comm...	Message	Author	Authored Date	Committer	Committed Date
	1.	PICK	84d7552	Reset a estado inicial	Javier Tuya	9 minutes ago	Javier Tuya	9 minutes ago
	2.	PICK	a5799b4	primer cambio en cambio-irebase	Javier Tuya	6 minutes ago	Javier Tuya	6 minutes ago
	3.	PICK	4c89f53	vuelta a tras, me habia equivocado	Javier Tuya	6 minutes ago	Javier Tuya	6 minutes ago
	4.	PICK	2e97b39	segundo cambio en cambio-irebase	Javier Tuya	5 minutes ago	Javier Tuya	5 minutes ago
	5.	PICK	6ae70a9	correccion segundo cambio	Javier Tuya	4 minutes ago	Javier Tuya	4 minutes ago

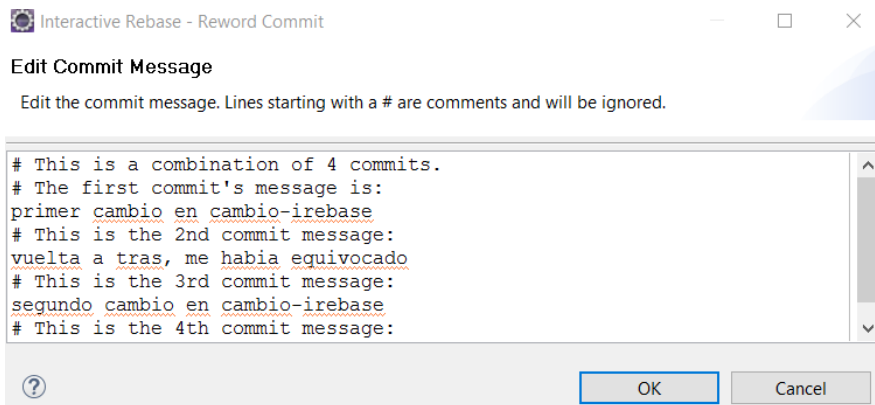
Ahora seleccionamos los commits que queremos modificar. En cada uno podemos indicar qué queremos hacer con él (*pick, edit...*). Lo que no interesa es hacer squash (juntar o fusionar commits en uno solo). para ello que seleccionamos los pasos 5, 4 y 3 y pulsamos **Squash**. **Notar que un squash junta un commit con el anterior**, por lo que el paso 2 no lo debemos seleccionar:



*Comits a fusionar*

Status	Step	Action	Comm...	Message	Author	Authored Date	Committer	Committed Date
	1.	PICK	84d7552	Reset a estado inicial	Javier Tuya	9 minutes ago	Javier Tuya	9 minutes ago
	2.	PICK	a5799b4	primer cambio en cambio-irebase	Javier Tuya	6 minutes ago	Javier Tuya	6 minutes ago
	3.	SQUASH	4c89f53	vuelta a tras, me habia equivocado	Javier Tuya	9 minutes ago	Javier Tuya	9 minutes ago
	4.	SQUASH	2e97b39	segundo cambio en cambio-irebase	Javier Tuya	8 minutes ago	Javier Tuya	8 minutes ago
	5.	SQUASH	6ae70a9	correccion segundo cambio	Javier Tuya	7 minutes ago	Javier Tuya	7 minutes ago

Finalmente, ejecutamos la operación con **Start** y nos saldrá la ventana donde indicaremos el mensaje del commit que reemplazará a estos cuatro.



Por defecto muestra todos los mensajes, aunque lo normal es que indiquemos un único mensaje que resuma todos los commits, p.e. "Consolidacion de 4 commits con rebase interactivo". Si vemos ahora la historia comprobamos como el ultimo commit ha sustituido a los anteriores:

Id	Message
d9e0c3a	<b>cambio-irebase</b> HEAD Consolidacion de 4 commits con rebase interactivo
84d7552	<b>master</b> origin/master Reset a estado inicial

Resultado  
única commit

Ahora podremos integrar estos cambios en master, en el que se reflejará solamente un commit en vez de cuatro.

NOTA: si los commit que habíamos hecho antes en esta rama fueron enviados al servidor con un push nos encontraremos con una situación no fast-forward porque Git intentará traer a nuestra rama los cambios que queremos remplazar por uno solo (porque los hemos juntado en el repositorio local, pero están separados en el remoto). Esto se evita con un *push force*, (opción no disponible en versiones antiguas de Eclipse). Se puede realizar desde la línea de comandos: **git push -f**.

### 7.3 Merge squash

En vez de juntar los commits en nuestra rama y luego hacer el merge de master con ella, se puede hacer el merge en master de forma que todos los cambios realizados en nuestra rama se vean como uno solo.

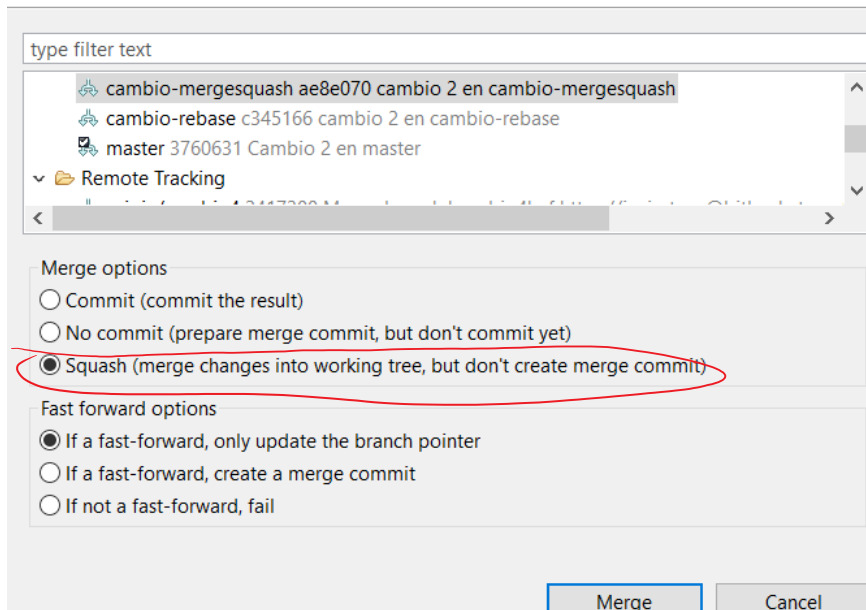
Supongamos la misma situación inicial que en la descripción de rebase anterior. Desde que nosotros hemos bifurcado a una rama (ahora denominada *cambio-mergesquash*) hemos realizado dos cambios en ella mientras que otros usuarios han realizado otros dos cambios en master. Nuestra rama (*cambio-mergesquash*) tendrá la siguiente historia:

Id	Message
83bb9c8	<b>cambio-mergesquash</b> HEAD cambio 2 en cambio-mergesquash
5c524dc	cambio 1 en cambio-mergesquash
ea21b71	<b>master</b> origin/master Reset a estado inicial

La operación *merge squash* permite incorporar en el workspace los cambios realizados en otra rama. De esta forma, podemos ir directamente a master (sin olvidar actualizarla previamente con pull) y hacer un merge squash con la rama *cambio-mergesquash*. Para ello basta con seleccionar **Team->Merge** y en el cuadro de diálogo que aparece indicar que se desea hacer Squash:

## Merge 'master'

Select a branch or tag to merge into the 'master' branch



Tras este merge, en el workspace tendremos incorporados los cambios realizados en la rama *cambio-mergesquash*. Después deberemos hacer un commit para que sean incorporados a la rama. El inconveniente de realizar *merge squash* es que se pierde completamente la relación entre las ramas que se están juntando, ya que, en este ejemplo, el efecto es como si hubiéramos realizado los cambios directamente sobre la rama *master*, que quedará sin ninguna relación con la rama *cambio-mergesquash* que es donde realmente habíamos realizado los cambios, por lo que no se usará en las prácticas.

## 7.4 Precauciones al alterar la historia de las ramas

Las operaciones descritas anteriormente utilizando rebase alteran la historia de las ramas, sustituyendo unos commit por otros. Por ello se debe tener siempre la siguiente precaución:

No realizar esas operaciones en ramas públicas (que han sido publicadas en el servidor remoto).

Alternativamente, si realizamos estas operaciones en una rama pública, debe asegurarse de que esta no es utilizada por ningún otro usuario. En particular, cuando se trabaja en equipo, nunca deben realizarse estas operaciones en la rama *master*.

De lo contrario podría darse la siguiente situación: Estamos en una rama (p.e. *master*) y hacemos un commit y push. Otro usuario crea una rama a partir de este commit. Si luego nosotros hacemos por ejemplo un rebase interactivo en el que juntamos este y otros commit, el efecto es que los commit que juntamos desaparecerán de Git y serán remplazados por uno nuevo. Ahora el otro usuario tiene una rama que ha sido creada desde un commit que ya no existe!!!

No olvidar que siempre que se hace rebase en una rama, si se realiza un push debe utilizarse la opción *force* (push -f).

## 7.5 Merge vs. Rebase

El procedimiento general para incorporar los cambios de nuestra rama en *master* es hacer un merge a *master*. Pero antes, debemos tener la precaución de incorporar en nuestra rama los cambios que hubieran podido ser realizados por otros usuarios en *master*, para lo cual tenemos la opción de hacer un merge o un rebase desde *master* hasta nuestra rama.

La ventaja de realizar rebase para ello es que mantendremos la historia de los cambios realizados por otros en master como un conjunto consecutivo de commits. Es como si hubiéramos empezado a incorporar cambios en nuestra rama en otro instante temporal posterior. Esto nos permite ver más fácilmente la historia de los cambios que hemos realizado.

Cuando se usa rebase para incorporar los cambios hechos en master por otros desarrolladores en nuestra rama de trabajo, es extremadamente importante recordar usar la opción *force* cuando se haga un push (`push -f`). La razón de esto es que en realidad cualquier operación rebase no elimina commits, sino que crea nuevos commits en nuestro repositorio local que cuando se envían a la rama remota alteran la historia.

Estos aspectos comentados en esta sección y en la anterior son discutidos en los siguientes enlaces, cuya lectura se aconseja:

- <https://www.atlassian.com/git/tutorials/merging-vs-rebasing>
- <https://www.freecodecamp.org/news/git-rebase-and-the-golden-rule-explained-70715eccc372/>

## 8 Otras posibilidades y características

1) Cuando estamos trabajando en una rama, y tenemos que ir hacia atrás a una versión de un commit anterior, se utiliza la el comando **Team->Reset** (normalmente indicando **Hard Reset**).

2) Sincronización: En Eclipse **Team-Synchronize** realiza una comparación del workspace con la última versión, permitiendo desde la misma vista realizar diversas operaciones (comparaciones, commit, etc.)

3) Cuando estamos trabajando en una rama y cambiamos a otra (p.e. para hacer un cambio de urgencia), si no hemos hecho un commit de los últimos cambios de la primera rama, perderemos estos cambios. Pero los cambios que tenemos pueden ser provisionales por lo que no deberíamos hacer un commit. Para estas situaciones se puede crear un nuevo workspace para trabajar en la nueva rama, o bien seguir en el mismo workspace y hacer (**Team->Stash**), asignando un nombre a los cambios que hemos hecho. Ahora podemos cambiar de forma segura a la otra rama y volver a recuperar los cambios de los que no se ha hecho commit en la rama inicial (nuevamente con **Team -> Stash**).

4) Precauciones con los finales de línea: Cuando se usan indistintamente sistemas Windows y Linux/Mac puede surgir problemas con los saltos de línea debido a los diferentes caracteres que usan (LF y CR los primeros, LF los segundos). De esta forma es posible que al hacer un merge de un fichero, aunque no haya habido cambios, todas las líneas parecen tener un cambio si los archivos de los que se está haciendo el merge tienen diferentes saltos de línea. Si esta es la situación, consultar antes la documentación de Git correspondiente: <https://help.github.com/articles/dealing-with-line-endings/>

5) Los repositorios como GitHub, BitBucket o GitLab permiten una forma de trabajo adicional mediante el denominado **Pull Request**, muy utilizado en el desarrollo de proyectos Open source. Esto permite que trabajemos en una bifurcación del repositorio maestro (fork) y que cuando tengamos unos cambios para incorporar a master, en vez de hacerlo nosotros mismos, realizamos un **Pull Request**. Los administradores o personas autorizadas en el repositorio maestro podrán entonces revisar y discutir los cambios para en su caso incorporarlos dicho repositorio.

6) Existen muchos otros comandos y variantes de los mismos. Ver documentación oficial de Git: <https://git-scm.com/documentation>. También es útil utilizar <http://stackoverflow.com/> para solucionar problemas particulares que puedan presentarse.

## 9 Organización del workflow

Organizar el workflow consiste en acordar cuál va a ser la forma de trabajar (cuántas ramas se utilizarán y para qué, cómo se realizarán los merge, si se utilizara rebase en vez de merge, etc.). [El workflow es lo primero que debe tener definido un equipo antes de usar Git, y debe estar particularizado a las características del equipo y del proyecto.](#)

[En cualquier proyecto en el que intervenga más de una persona deben utilizarse ramas para el desarrollo en paralelo \(nunca realizar ningún tipo de trabajo en master\).](#)

En un proyecto ágil de pequeño tamaño con un desarrollo continuo como los de las prácticas de laboratorio, la forma más simple de organizarse es:

- Disponer de la rama *master* para integrar los cambios de todos los usuarios, pero nunca desarrollar en ella.
- Disponer de una rama privada por cada una de las historias de usuario que se van a desarrollar. Alternativamente, se podría tener una rama privada por desarrollador, aunque se necesitarán otras ramas si en algún momento hay que volver hacia otra tarea no correspondiente a la historia que se está implementando en este momento. **NOTA: En las prácticas de laboratorio se exigirá utilizar una rama por historia de usuario.**
- Antes de integrar en master cualquier cambio realizado en la una rama de trabajo:
  - Usar rebase interactivo con squash para eliminar commits no relevantes (si no, pasarían finalmente a master).
  - Integrar los cambios que puedan haber realizado en master otros usuarios (con merge o rebase) y probar el conjunto desde la rama de trabajo
- Finalmente, desde master, hacer el merge para incorporar los cambios de la rama de trabajo (nunca rebase).

[Adicionalmente, puede utilizarse también una rama adicional denominada \*develop\*.](#) Los cambios se integrarán en *develop* y de ahí a master. Esto permite tener una mayor estabilidad de la versión existente en master. [En la rama \*develop\* se realizarán las pruebas de integración del trabajo realizado en el resto de ramas](#) (sigue siendo exigido que cada desarrollador pruebe el trabajo en su propia rama). En ella se realizarán las últimas correcciones antes de integrar los cambios en *master*.

A modo de información adicional, indicar que existen diferentes formas de organizar el workflow (p.e. el *Forking workflow* que implica el uso de *pull request* para revisar los cambios). Uno de los más conocidos (aunque también complejo) el denominado *GitFlow*, que se ilustra en la siguiente figura. Puede verse más información sobre diferentes workflows en: <https://www.atlassian.com/git/tutorials/comparing-workflows/gitflow-workflow>

